User Interface Programming with Python

Peter Hartford

Table of Contents

This should teach all of the basics

Fact of the day: some people say “Gee u eye”,
Peter says “Gooey” because it sounds funnier

When should you create a user interface?

- User interfaces are a somewhat large undertaking
  - Can easily become a waste of time
  - Sometimes better to just have a script

Instead of asking whether you can (you can), ask whether you should…

When should you create a user interface?

- User interfaces are a somewhat large undertaking
  - Can easily become a waste of time
  - Sometimes better to just have a script
- Ask yourself
  - Am I using this script often (e.g. recording data)? ✓
  - Am I repeatedly changing parameters and adjusting things? ✓
  - Am I continuously changing the functions? ✗
  - Is someone going to use your code? ✓
  - Is someone going to **edit** your code? ✗
- Post processing scripts usually don't need a GUI

Instead of asking whether you can (you can), ask whether you should…

Part 1: File Structure/Preliminary GUI

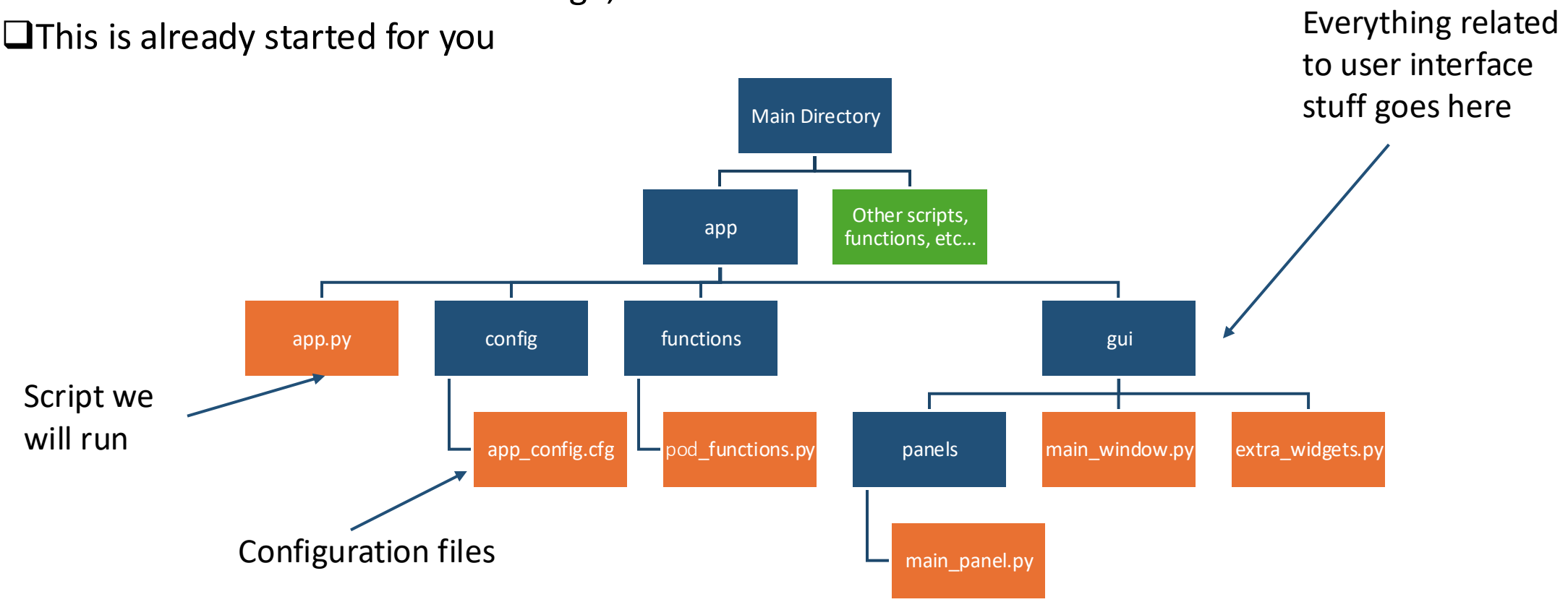
Example Script: POD Filter for PIV Images

- Why is this a good example?
  - Has a few parameters that need to be changed
  - Can save and load images
  - The script takes time to run (more on this later)…
  - Ability to extend to interactive plotting

```python
# -*- coding: utf-8 -*-
"""
Created on Thu Sep 26 11:37:58 2019

@author: mendez, torres
"""

import os  # This is to understand which separator in the paths (/ or \)

import matplotlib.image as mpimg
import numpy as np  # This is for doing math
from skimage.io import imread, imshow, imsave  # this is for Matlab users

def generate_filename(folder, number, pair, pic_format):
    """
    simple function to generate filenames
    :param folder: where to look or save
    :param number: number of the picture
    :param pair: a or b
    :param pic_format: format of the image
    :return:
    """
    return folder + os.sep + 'A'+'%03d' % (number + 1) + pair + '.' + pic_format


# Image Cropping, Flipping and pre-processing using the 1POD mode removal
# Fore more advanced version, see https://seis.bristol.ac.uk/~aexrt/PIVPODPreprocessing/
# To do list: Implement Frequency based filter.

# Ensure current working directory
cwd = os.path.dirname(os.path.realpath(__file__))
os.chdir(cwd) #chdir used for change directory
print(os.getcwd())

# Folder in
FOL_IN = 'RAW_IMAGES'
# Processing Images
FOL_OUT = 'Pre_Pro_PIV_IMAGES'  # Where will the result be
if not os.path.exists(FOL_OUT):
    os.mkdir(FOL_OUT)


# To define the crop area you will usually need to load at least one image
# and plot on it a rectangle
Num = 10  # This is the number of the image
Name=generate_filename(FOL_IN, 10, 'a', 'tif')

Im = imread(Name, as_gray=True)
ny, nx = Im.shape
```

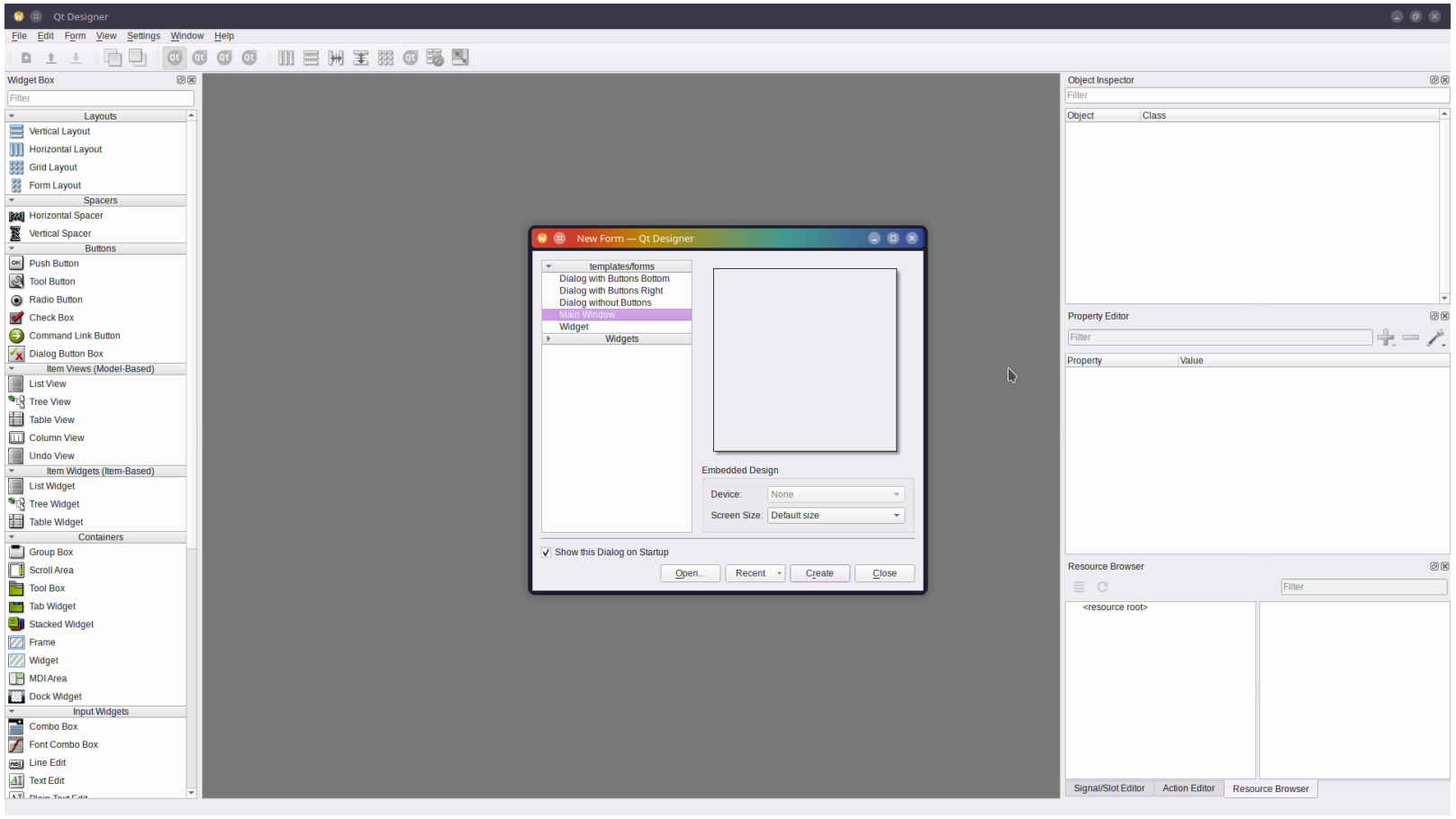
Folder/File Structure

- File organization is key since we are creating multiple python files
- Usually use something like this
  - The files we will create are in orange, folders in blue
  - This is already started for you

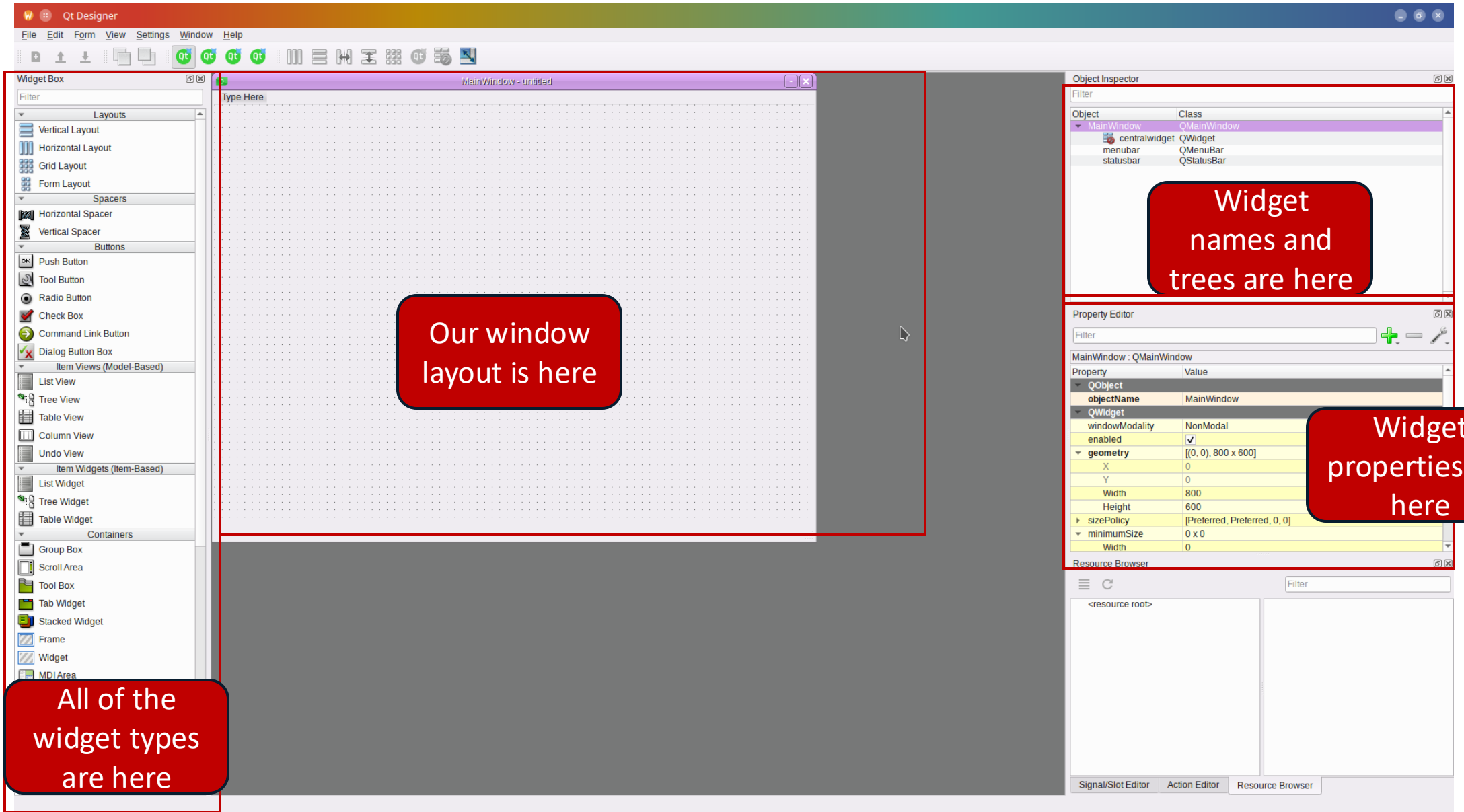


Getting Started: Qt Designer

- First thing we will generate is a panel file
- Open Qt Designer
  - In the terminal type “pyqt5-tools designer”
  - Qt Designer should open with a new form dialog
  - Create a "main window" form



Navigating QT Designer



Setting up the main window

- Start by resizing the main window and changing the title
  - Resize the main window to 1000x800 – gives us space to work with
  - The title is also nice to change for reference (you can also add an icon, etc..., etc...)

MainWindow : QMainWindow

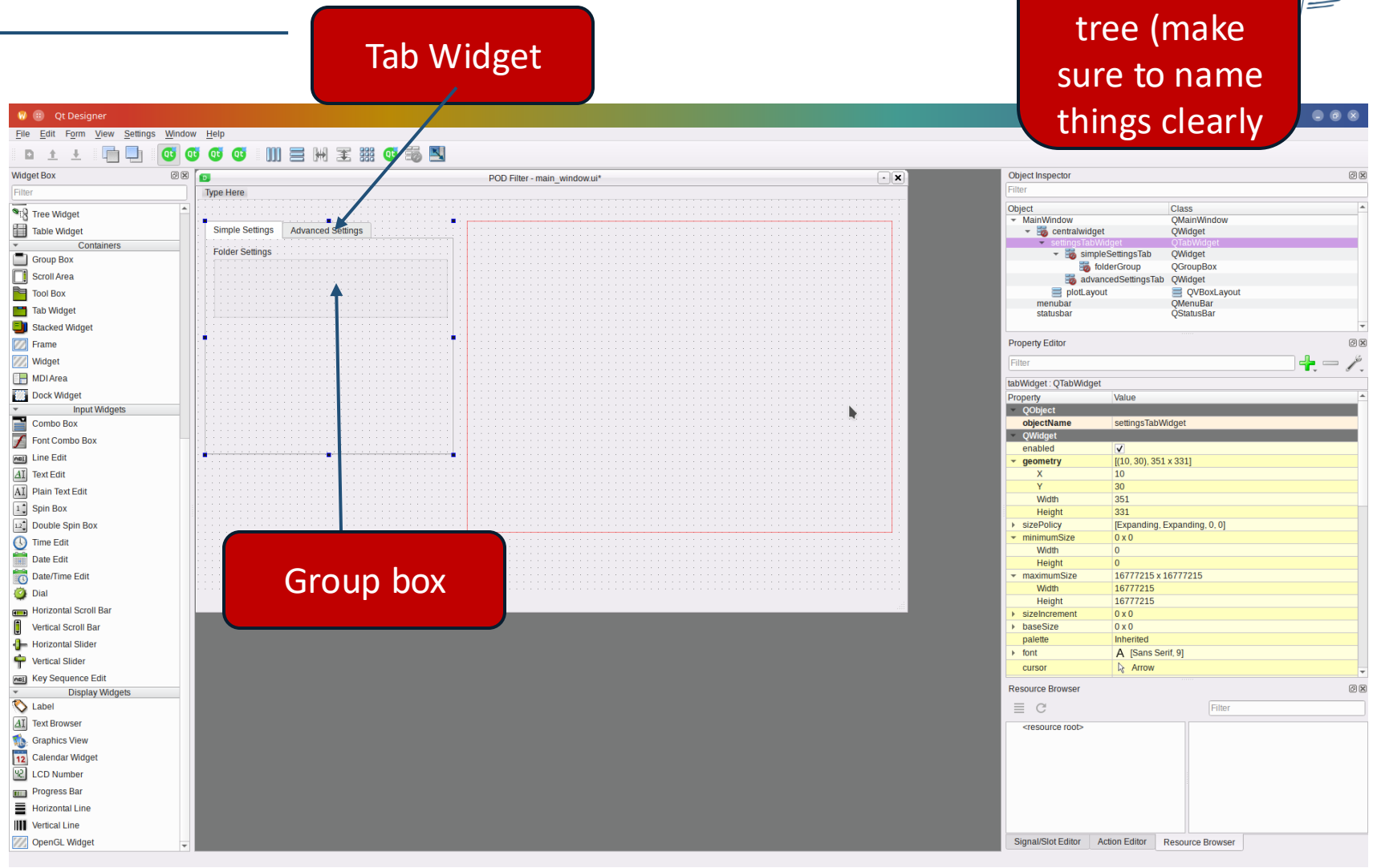
Property	Value
QObject	
objectName	MainWindow
QWidget	
windowModality	NonModal
enabled	☑
geometry	[(0, 0), 1000 x 800]
X	0
Y	0
Width	1000
Height	800
sizePolicy	[Preferred, Preferred, 0, 0]
minimumSize	0 x 0
Width	0

MainWindow : QMainWindow

Property	Value
mouseTracking	☐
tabletTracking	☐
focusPolicy	NoFocus
contextMenuPolicy	DefaultContextMenu
acceptDrops	☐
windowTitle	MainWindow
windowIcon	D
Theme	
Normal Off	D
Normal On	D
Disabled Off	D
Disabled On	D

Adding Widgets

- We are going to start by adding organization widgets
  - Putting widgets in groups helps with organization and user readability
  - The tab widget creates tabs (like a web browser) for widgets



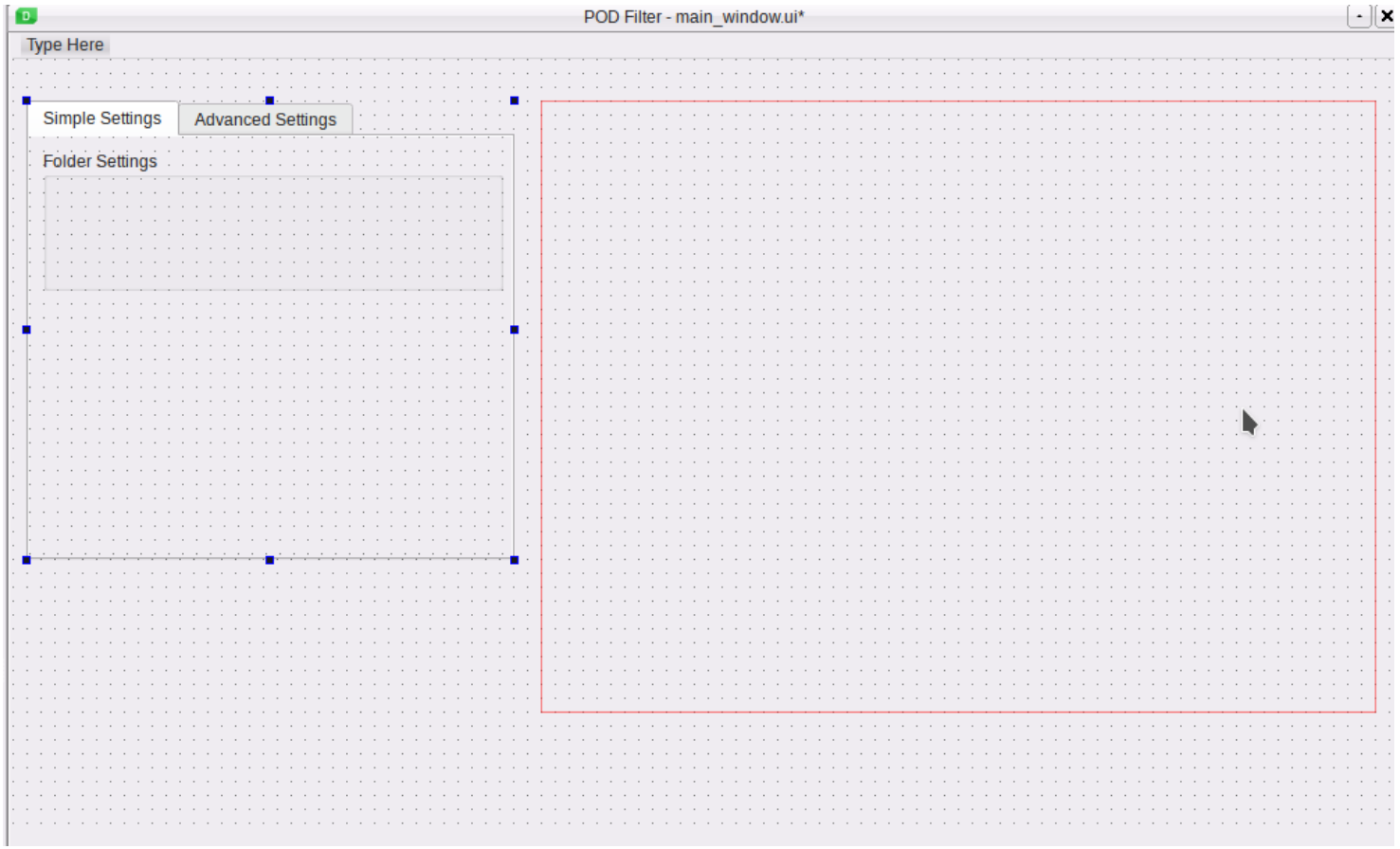
Editing Tab Text

- Tab titles need to be edited here (not so obvious)
  - Click the tab you want to name
  - Change the "currentTabText" to change the title

tabWidget : QTabWidget	
Property	Value
layoutDirection	LeftToRight
autoFillBackground	☐
styleSheet	
locale	English, United States
inputMethodHints	ImhNone
QTabWidget	
tabPosition	North
tabShape	Rounded
currentIndex	0
iconSize	16 x 16
elideMode	ElideNone
usesScrollButtons	☑
documentMode	☐
tabsClosable	☐
movable	☐
tabBarAutoHide	
currentTabText	Simple Settings
currentTabName	simpleSettingsTab
currentTabIcon	
currentTabToolTip	
currentTabWhatsThis	

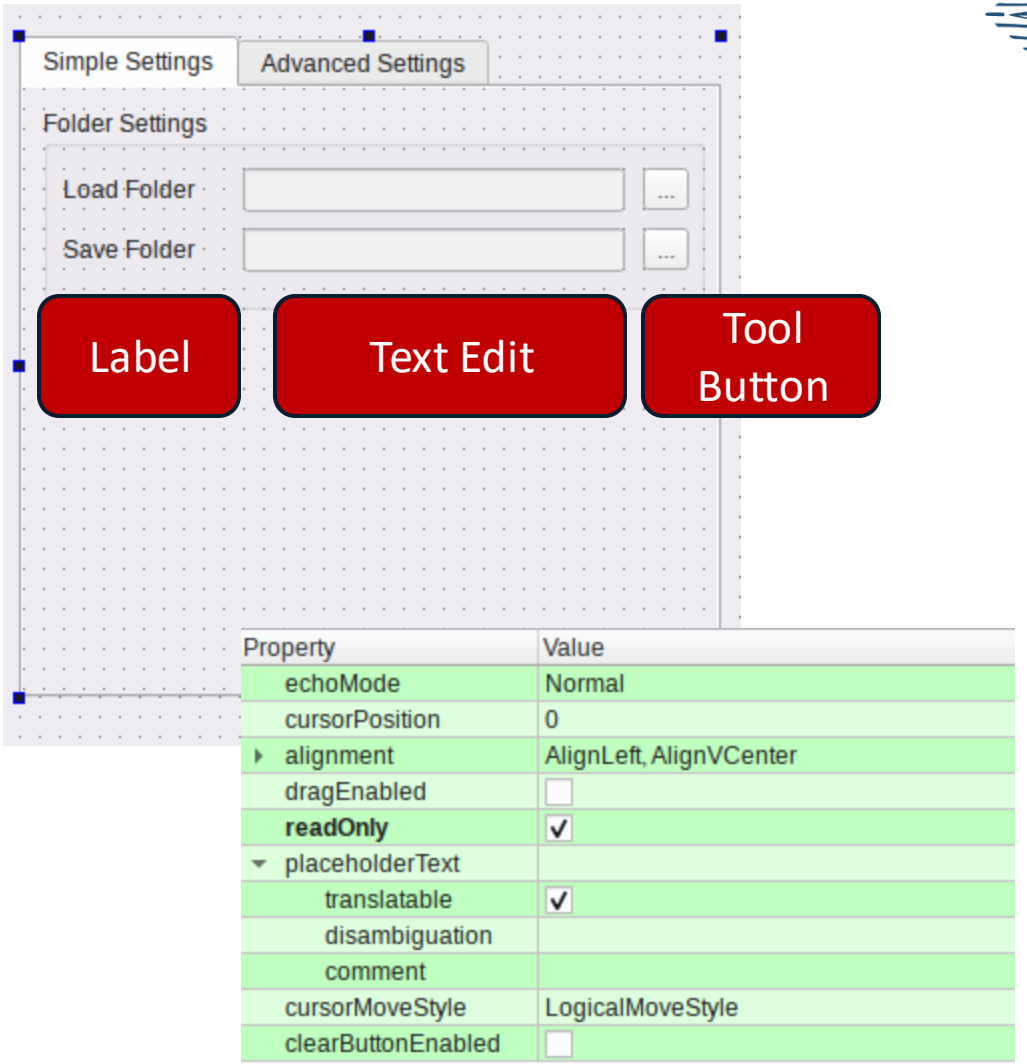
Plot Layout

- This needs to be added for our custom matplotlib widget
- (Red rectangle)



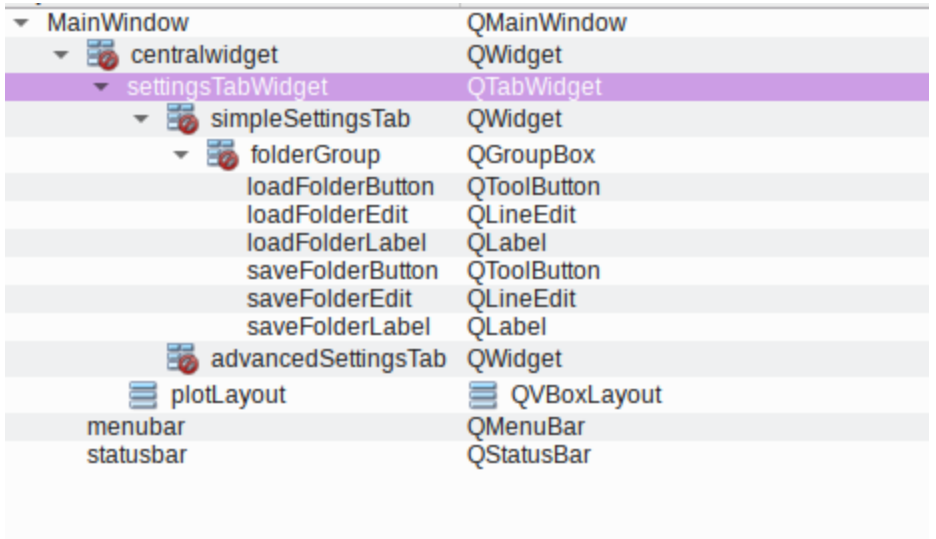
Adding **More** Widgets

- Let's populate the settings group with some more widgets
- A Label Widget lets the user know what they're looking at
- A text widget is useful for input/output
  - For these we want to make the box "readOnly"
- The Tool Button will be used to select our folders



Naming Widgets

- It is **very** important to name your widgets properly
  - Good name: "recordButton," "flipImageCheckbox"
  - Bad name: "petersFancyWidgetThatDoesStuff," "widget56"
- When creating scripts, it should be obvious what the widget does
- The name should have
  - Functionality indicator
  - What type of widget it is



Exporting to a python file

- We have our initial app layout – save and create a .ui file
- To export to a python file run this command
  - "pyuic5 /path/to/ui/file.ui -o /path/to/python/ui/file.py"
  - This generates a file with all of the layouts
- **Do not edit this file**
  - It will get rewritten if you export the layout again

main_window.py | main_window_gui.py | app.py | extra_widgets.py

```python
# -*- coding: utf-8 -*-

# Form implementation generated from reading ui file 'main_window.ui'
#
# Created by: PyQt5 UI code generator 5.15.7
#
# WARNING: Any manual changes made to this file will be lost when pyuic5 is
# run again.  Do not edit this file unless you know what you are doing.


from PyQt5 import QtCore, QtGui, QtWidgets


class Ui_MainWindow(object):
    def setupUi(self, MainWindow):
        MainWindow.setObjectName("MainWindow")
        MainWindow.resize(1000, 600)
        self.centralwidget = QtWidgets.QWidget(MainWindow)
        self.centralwidget.setObjectName("centralwidget")
        self.tabWidget = QtWidgets.QTabWidget(self.centralwidget)
        self.tabWidget.setGeometry(QtCore.QRect(10, 30, 351, 331))
        self.tabWidget.setObjectName("tabWidget")
        self.simpleSettingsTab = QtWidgets.QWidget()
        self.simpleSettingsTab.setObjectName("simpleSettingsTab")
        self.folderGroup = QtWidgets.QGroupBox(self.simpleSettingsTab)
        self.folderGroup.setGeometry(QtCore.QRect(10, 10, 331, 101))
        self.folderGroup.setObjectName("folderGroup")
        self.loadFolderLabel = QtWidgets.QLabel(self.folderGroup)
        self.loadFolderLabel.setGeometry(QtCore.QRect(10, 30, 91, 21))
        self.loadFolderLabel.setObjectName("loadFolderLabel")
        self.loadFolderEdit = QtWidgets.QLineEdit(self.folderGroup)
        self.loadFolderEdit.setEnabled(False)
        self.loadFolderEdit.setGeometry(QtCore.QRect(100, 30, 191, 22))
        self.loadFolderEdit.setObjectName("loadFolderEdit")
        self.loadFolderButton = QtWidgets.QToolButton(self.folderGroup)
        self.loadFolderButton.setGeometry(QtCore.QRect(300, 30, 23, 21))
        self.loadFolderButton.setObjectName("loadFolderButton")
        self.saveFolderLabel = QtWidgets.QLabel(self.folderGroup)
        self.saveFolderLabel.setGeometry(QtCore.QRect(10, 60, 91, 21))
        self.saveFolderLabel.setObjectName("saveFolderLabel")
        self.saveFolderEdit = QtWidgets.QLineEdit(self.folderGroup)
        self.saveFolderEdit.setEnabled(False)
        self.saveFolderEdit.setGeometry(QtCore.QRect(100, 60, 191, 22))
        self.saveFolderEdit.setObjectName("saveFolderEdit")
        self.saveFolderButton = QtWidgets.QToolButton(self.folderGroup)
        self.saveFolderButton.setGeometry(QtCore.QRect(300, 60, 23, 21))
        self.saveFolderButton.setObjectName("saveFolderButton")
```

Creating our Main Window File

- The exported file creates a class – we want to import this into a separate file
  - If we change the UI file, it will completely rewrite our layout script
  - "Separate the GUI and Business logic"
- Every function will be initialized here
  - A separate class will be created to contain the POD filter functions
  - This helps to keep files small and organized

main_window.py | main_window_gui.py | app.py | extra_widgets.py

```
@author: peter
"""

#Import the matplotlib widget and main window
from panels.main_window import Ui_MainWindow
from gui.extra_widgets import MplWidget

#Function Classes - we use this class for all of our functions
#from functions.functions_pod import PODWrapper

#Imports for pyqt5 widgets
from PyQt5 import QtWidgets, QtCore, QtGui
from PyQt5.QtCore import pyqtSlot #This is for threading/signals

#Dark theme - automatically applies nice stylesheet
#import qdarktheme

#For configuration files
from configparser import ConfigParser
import numpy as np
from pathlib import Path

import matplotlib
matplotlib.use('Qt5Agg')


#Main Window Object
class MainWindow(Ui_MainWindow):
    #Init function (runs on creation)
    def __init__(self):
        #Inherit objects from our panel file
        super(MainWindow, self).__init__()

        #This is important for setting window properties
        self.window = QtWidgets.QMainWindow()

        #This does all of the layout and attaches widgets to the window object
        self.setupUi(self.window)


        #Create our plot widget and attach to the plot layout
        self.livePlotWidget = MplWidget(navigationToolbar = True)
        self.plotLayout.addWidget(self.livePlotWidget)

        #Set load folder to default
        self.loadFolder = Path('/')
```

Import our class here

Create new class and inherit

Embedding a matplotlib plot

- This will be useful later
- The matplotlib widget class (custom made) gets embedded into the GUI
- Enables full functionality of matplotlib (quite useful)

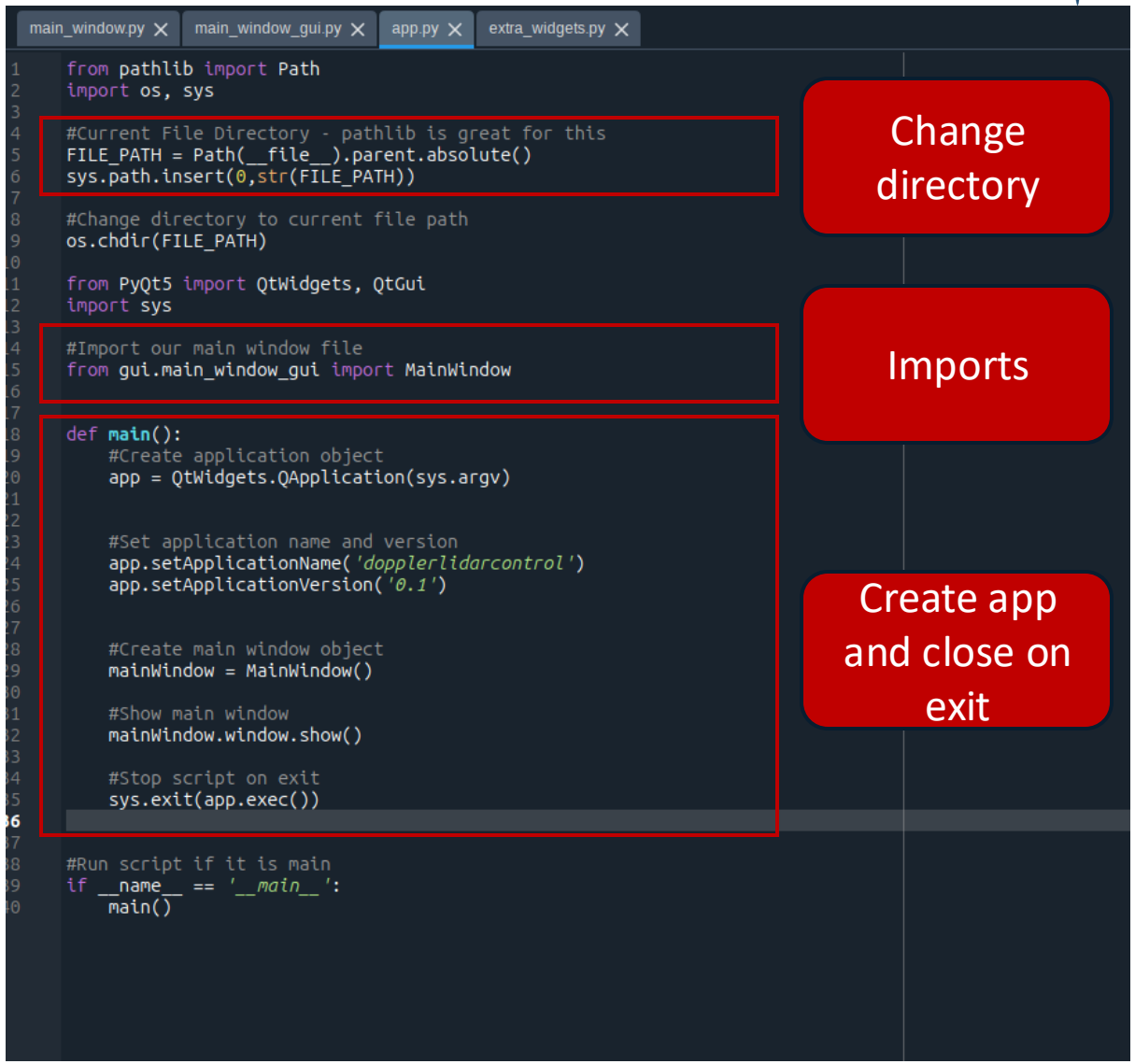
main_window.py | main_window_gui.py | app.py | extra_widgets.py

```
@author: peter
"""

#Import the matplotlib widget and main window
from panels.main_window import Ui_MainWindow
from gui.extra_widgets import MplWidget

#Function Classes - we use this class for all of our functions
#from functions.functions_pod import PODWrapper

#Imports for pyqt5 widgets
from PyQt5 import QtWidgets, QtCore, QtGui
from PyQt5.QtCore import pyqtSlot #This is for threading/signals

#Dark theme - automatically applies nice stylesheet
#import qdarktheme

#For configuration files
from configparser import ConfigParser
import numpy as np
from pathlib import Path

import matplotlib
matplotlib.use('Qt5Agg')


#Main Window Object
class MainWindow(Ui_MainWindow):
    #Init function (runs on creation)
    def __init__(self):
        #Inherit objects from our panel file
        super(MainWindow, self).__init__()

        #This is important for setting window properties
        self.window = QtWidgets.QMainWindow()

        #This does all of the layout and attaches widgets to the window object
        self.setupUi(self.window)


        #Create our plot widget and attach to the plot layout
        self.livePlotWidget = MplWidget(navigationToolbar = True)
        self.plotLayout.addWidget(self.livePlotWidget)

        #Set load folder to default
        self.loadFolder = Path('/')
```

Embed matplotlib widget

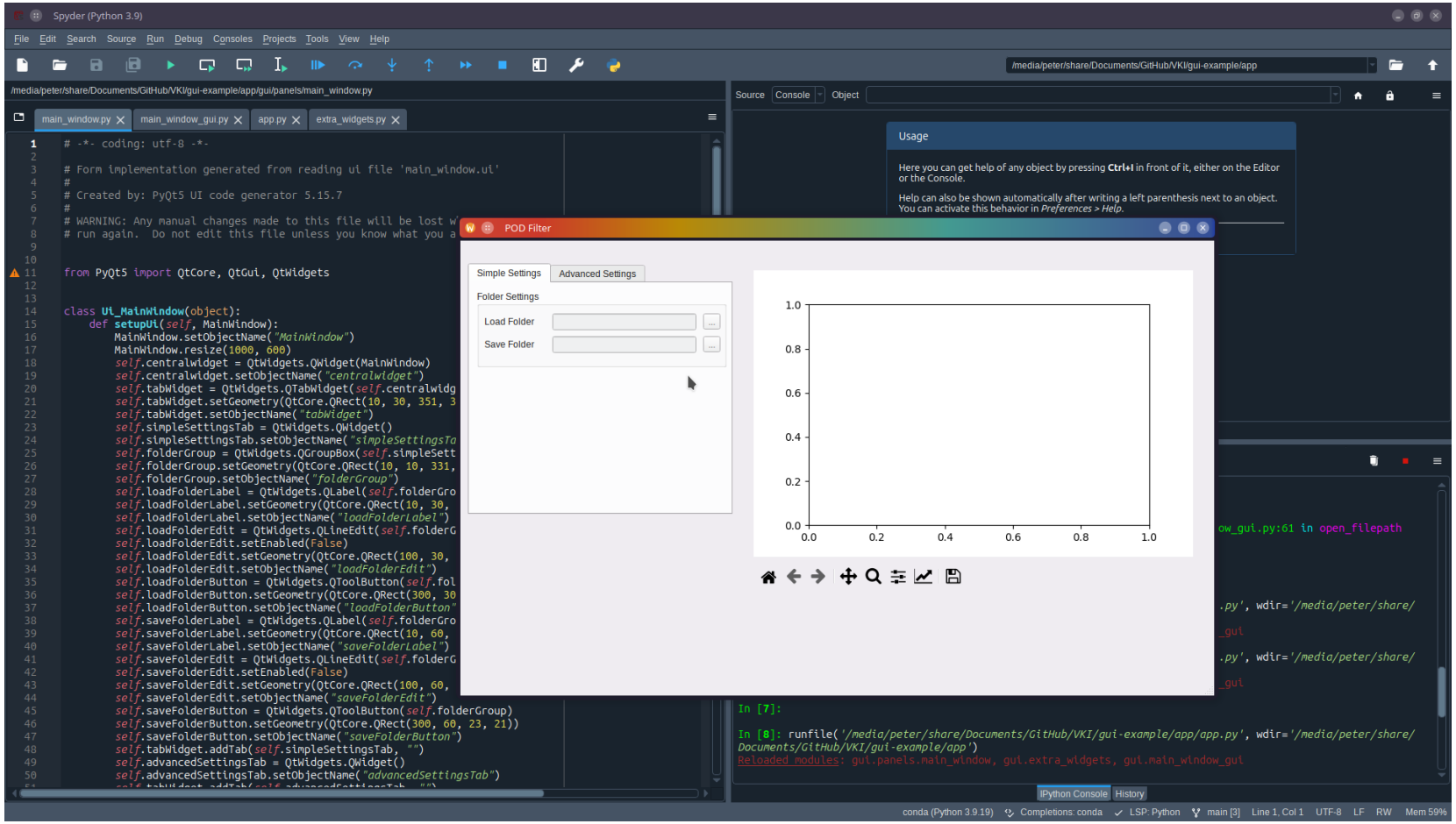
What app.py does

- app.py is the script that we run – it's good to keep things short and simple
- It has 3 major components
  - Import our custom classes
  - Set the working directory (helps prevent import issues)
  - Create the application instance and tell python to close when the app stops



Running app.py

- Spyder/vs code can run the same app.py
  - When this is run, a new window should appear!



Part 2: Our First Function

Our first function: Selecting an Image Folder

- One of the most useful things for GUIs is to create an easy workflow for commonly used scripts
- Selecting a folder for your workspace is always useful to do
- Pyqt has a built-in function for this

```
#These two functions are basically the same - just for saving/loading...
def open_filepath(self, var):
    #Create a file dialog to open a new folder - return it with a path object - default directory is record folder
    self.loadFolder = Path(QtWidgets.QFileDialog.getExistingDirectory(directory = str(self.loadFolder)))

    #Set the text to the file path
    self.loadFolderEdit.setText(str(self.loadFolder))


def save_filepath(self):
    #Create a file dialog to open a new folder - return it with a path object - default directory is record folder
    self.saveFolder = Path(QtWidgets.QFileDialog.getExistingDirectory(directory = str(self.loadFolder)))

    #Set the text to the file path
    self.saveFolderEdit.setText(str(self.saveFolder))
```

Connecting functions

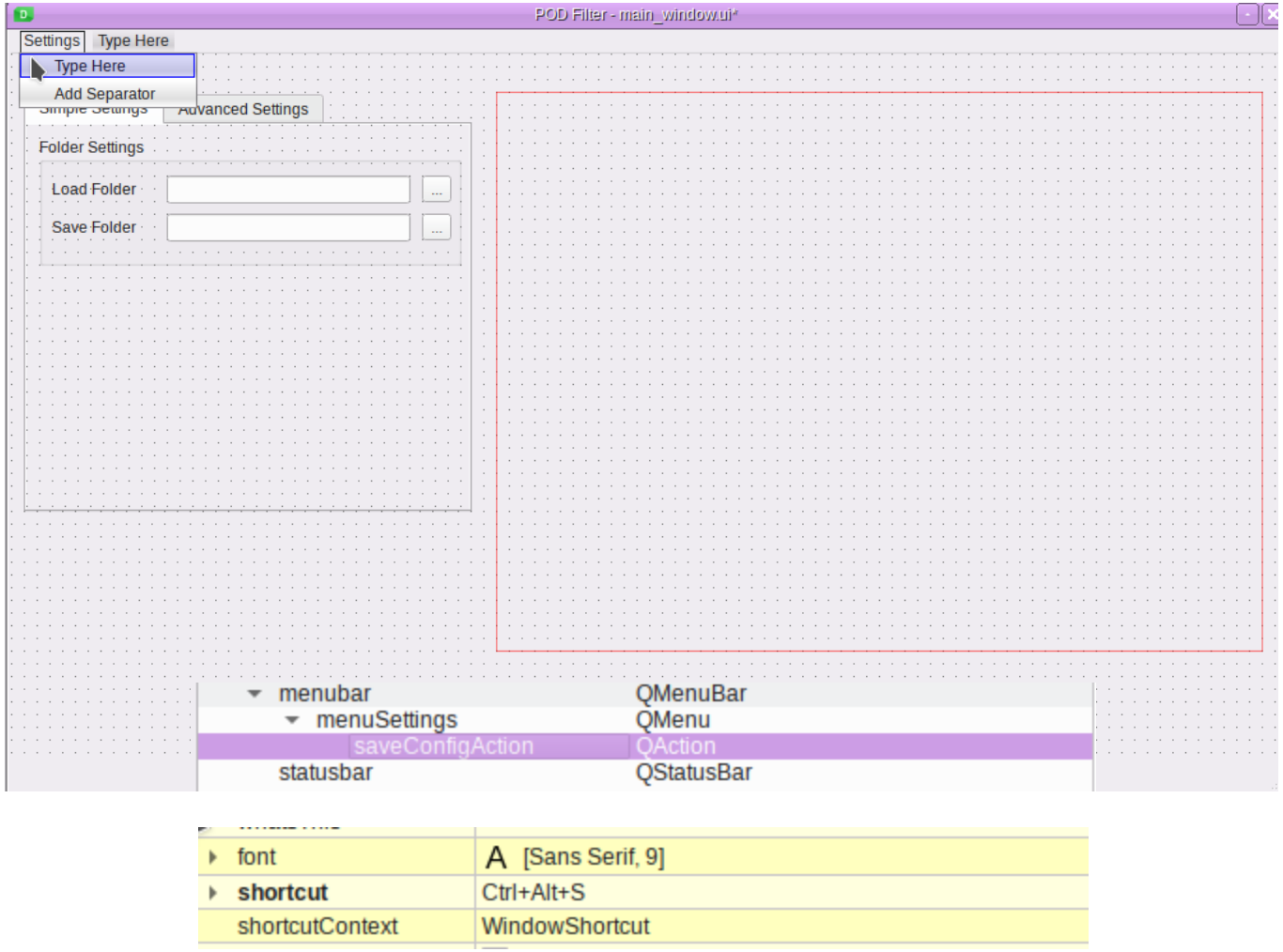
- The functions need to be connected to the widget that performs the action
- This is quite simple to do!
  - You usually connect the widget action to the function
  - If there's an input variable use the "lambda" functionality
- Run app.py again and the button should open a file dialog

```python
#Main Window Object
class MainWindow(Ui_MainWindow):
    #Init function (runs on creation)
    def __init__(self):
        #Inherit objects from our panel file
        super(MainWindow, self).__init__()

        #This is important for setting window properties
        self.window = QtWidgets.QMainWindow()

        #This does all of the layout and attaches widgets to the window
        self.setupUi(self.window)


        #Create our plot widget and attach to the plot layout
        self.livePlotWidget = MplWidget(navigationToolbar = True)
        self.plotLayout.addWidget(self.livePlotWidget)

        #Set load folder to default - this will be changed when we create a config file
        self.loadFolder = Path('/')

        #Connect buttons to functions
        self.loadFolderButton.clicked.connect(lambda: self.open_filepath(1))
        self.saveFolderButton.clicked.connect(self.save_filepath)
```

lambda: functionality

Creating a Settings File/Titlebar

- A settings file is useful in the event that your GUI closes/crashes
- Usually this can go on the titlebar of the GUI
- Create a menu entry called Save Config
- We can also add a keyboard shortcut
  - i.e. "ctrl+alt+s"



Saving a Settings File

- Now let's create a function for this
- A very useful package "configparser" enables us to create readable configuration files
- Remember to connect the button to the function (or it won't run)

```python
#This function saves default settings
    def save_app_config(self):
        #Create a config parser object for the settings
        self.settings = ConfigParser()

        #Folder Settings create a nested dictionary
        self.settings['Folder Settings'] = {}

        #Now lets get the value of the folder boxes
        self.settings['Folder Settings']['LoadFolder'] = self.loadFolderEdit.text()
        self.settings['Folder Settings']['SaveFolder'] = self.saveFolderEdit.text()

        #Save the write the settings to our config file
        with open(CONFIG_PATH, 'w') as settings_file:
            self.settings.write(settings_file)
```

main_window.py | main_window_gui.py* | app.py | extra_widgets.py | app_config.cfg*

```
[Folder Settings]
loadfolder = /media/peter/share/Documents/Data/PIV Data/100mus
savefolder =
```

```python
class MainWindow(Ui_MainWindow):
    #Init function (runs on creation)
    def __init__(self):
        #Inherit objects from our panel file
        super(MainWindow, self).__init__()

        #This is important for setting window properties
        self.window = QtWidgets.QMainWindow()

        #This does all of the layout and attaches widgets to the window object
        self.setupUi(self.window)


        #Create our plot widget and attach to the plot layout
        self.livePlotWidget = MplWidget(navigationToolbar = True)
        self.plotLayout.addWidget(self.livePlotWidget)

        #Set load folder to default - this will be changed when we create a config file
        self.loadFolder = Path('/')

        #Connect buttons to functions
        self.loadFolderButton.clicked.connect(lambda: self.open_filepath(1))
        self.saveFolderButton.clicked.connect(self.save_filepath)

        #Connect our status bar actions to functions
        self.saveConfigAction.triggered.connect(self.save_app_config)

        #Automatically load our app configuration
        self.load_app_config()
```

Loading a Settings File

- We can make the GUI automatically load the parameters we save every time we run it
- Also can connect to a button

```
class MainWindow(Ui_MainWindow):
    #Init function (runs on creation)
    def __init__(self):
        #Inherit objects from our panel file
        super(MainWindow, self).__init__()

        #This is important for setting window properties
        self.window = QtWidgets.QMainWindow()

        #This does all of the layout and attaches widgets to the window object
        self.setupUi(self.window)


        #Create our plot widget and attach to the plot layout
        self.livePlotWidget = MplWidget(navigationToolbar = True)
        self.plotLayout.addWidget(self.livePlotWidget)

        #Set load folder to default - this will be changed when we create a config file
        self.loadFolder = Path('/')

        #Connect buttons to functions
        self.loadFolderButton.clicked.connect(lambda: self.open_filepath(1))
        self.saveFolderButton.clicked.connect(self.save_filepath)

        #Connect our status bar actions to functions
        self.saveConfigAction.triggered.connect(self.save_app_config)

        #Automatically load our app configuration
        self.load_app_config()
```

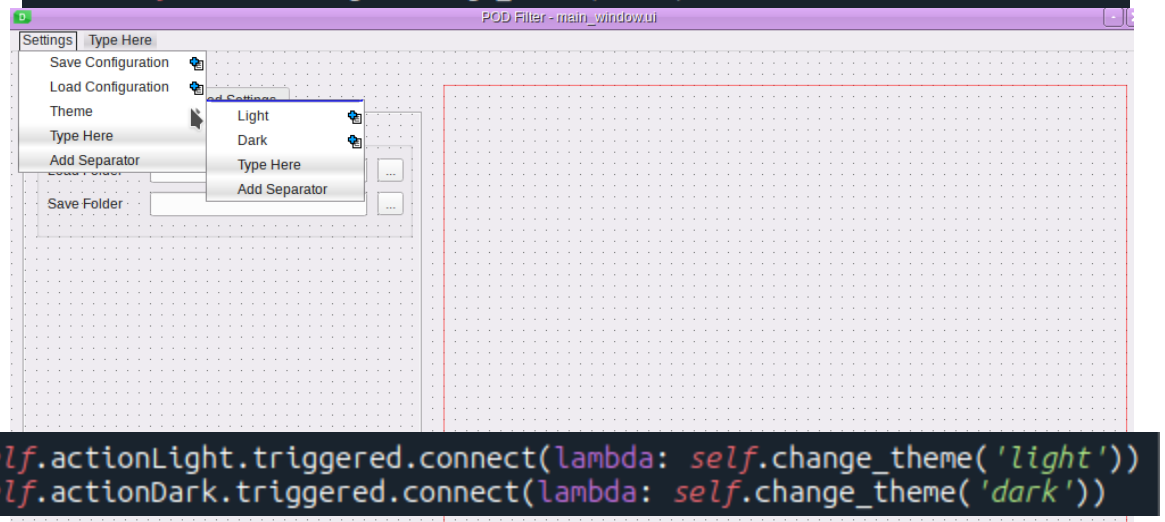
```
    #This function loads default settings
    def load_app_config(self):
        #Create config parser object
        self.settings = ConfigParser()

        #Check if the config file exists
        if CONFIG_PATH.exists():
            #Read config file
            self.settings.read(CONFIG_PATH)

            #Set the widget values
            self.set_settings()

        #If the file doesn't exist, nothing will load :(

    def set_settings(self):
        #Lets set the widget text to our settings file
        self.loadFolderEdit.setText(self.settings['Folder Settings']['LoadFolder'])
        self.saveFolderEdit.setText(self.settings['Folder Settings']['SaveFolder'])
```

Just call the function when the GUI gets loaded!

Changing Themes

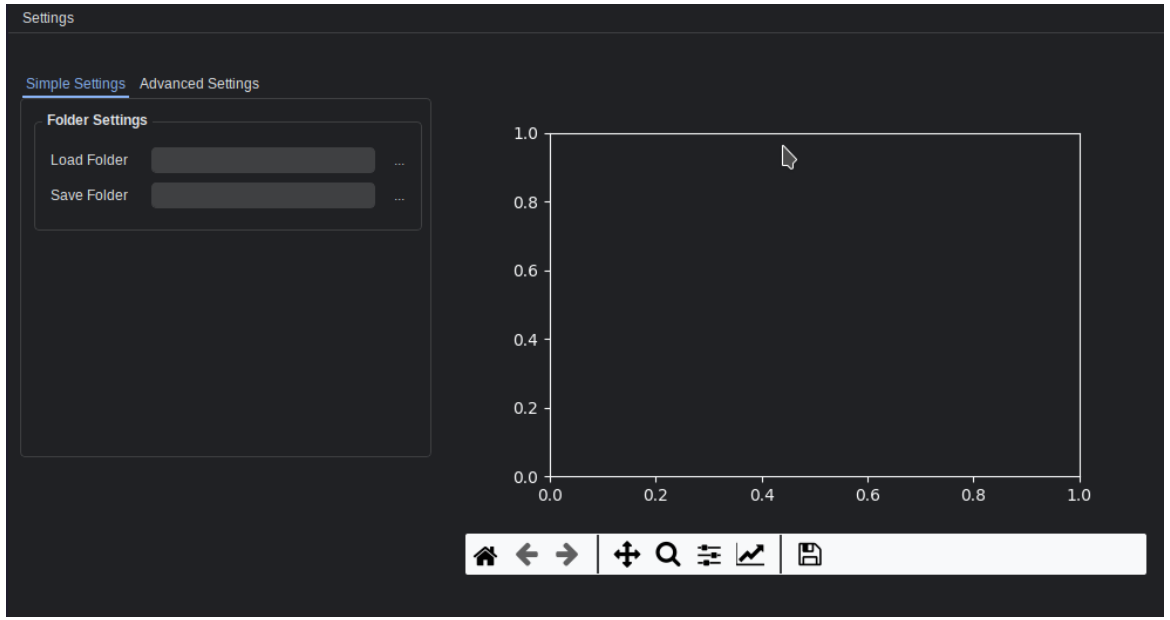
- This is extra, but there is a useful package that can easily make the GUI look good everywhere
  - Besides, who doesn't like dark theme
- This goes nicely in the titlebar
- Automatically load the theme by adding it to the config

```
#Quick theme changer - makes everything look good at once
def change_theme(self, theme):
    qdarktheme.setup_theme(theme)

    #Change theme for plots
    self.livePlotWidget.change_theme(theme)
```



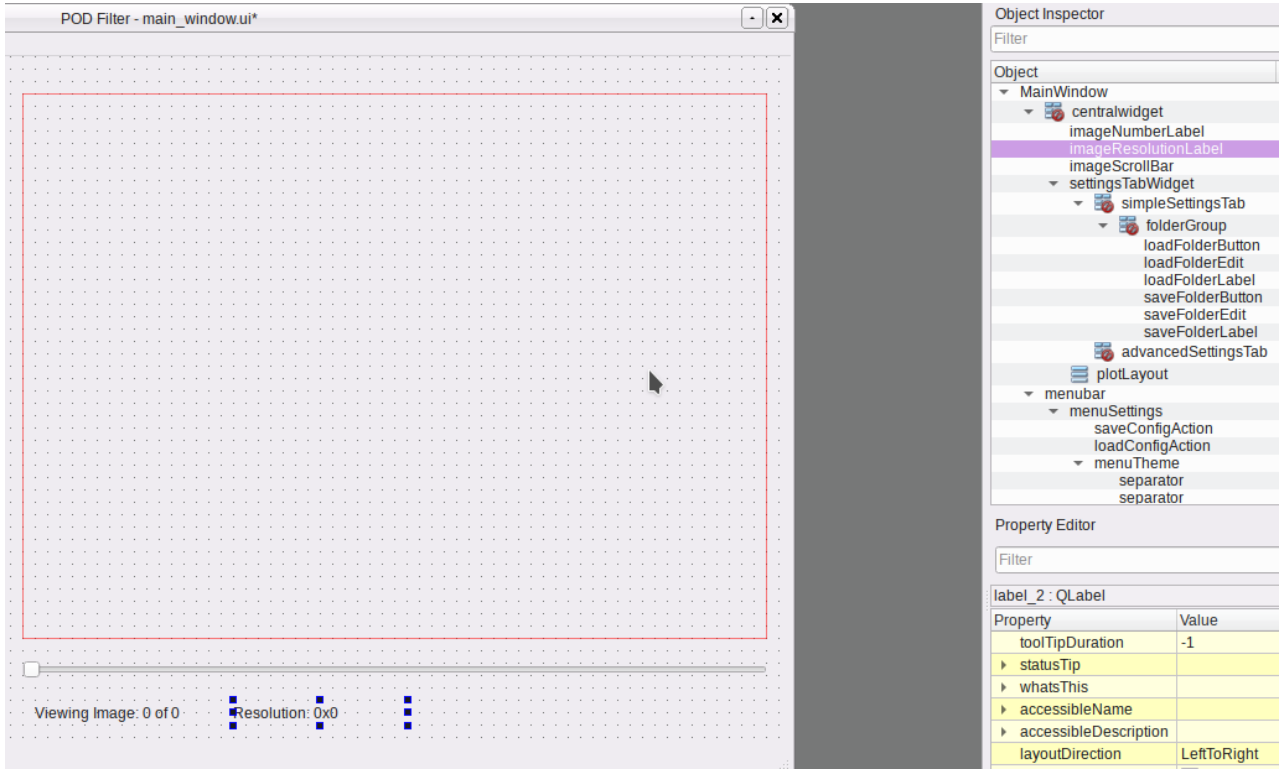

```
self.actionLight.triggered.connect(lambda: self.change_theme('light'))
self.actionDark.triggered.connect(lambda: self.change_theme('dark'))
```

Add these buttons and connect



Looking at Images

- Now let's set up our matplotlib widget!
- Our goal is to inspect the images **before** and **after** the POD
  - This is a big undertaking, but we can start small
  - Let's first create a tool to view the images in our folder
  - This function can be run **without threading** because loading an image won't take long
- Let's add a slider and a couple labels below our plot
- Set all of them to "disabled"



Looking at Images

- Now let's create our functions
  - First we need to figure out if there are image files present in the folder
  - We create a sorted list of them
- Then we can update the widgets accordingly
  - Set the maximum scroll bar value to the image list length

```
#This tells the app the image files in our folder
def get_images(self):
    #List all files in the image folder and sort
    self.imageList = sorted([file for file in self.loadFolder.glob('*.tif')])

    #Update widgets if there are images
    if len(self.imageList) > 0:
        #Set the scroll bar maximum
        self.imageScrollBar.setMaximum(len(self.imageList))

        #Enable the widgets
        self.imageScrollBar.setEnabled(True)
        self.imageNumberLabel.setEnabled(True)

        #Set the label text
        self.imageNumberLabel.setText('Image Number: %i of %i'%(0, len(self.imageList)))

        #Update with the initial image
        self.update_image(0)

    #Disable the widgets if there arent any images
    else:
        #This can be simplified if they're all in the same group
        self.imageScrollBar.setDisabled(True)
        self.imageNumberLabel.setDisabled(True)
        self.imageResolutionLabel.setDisabled(True)
```

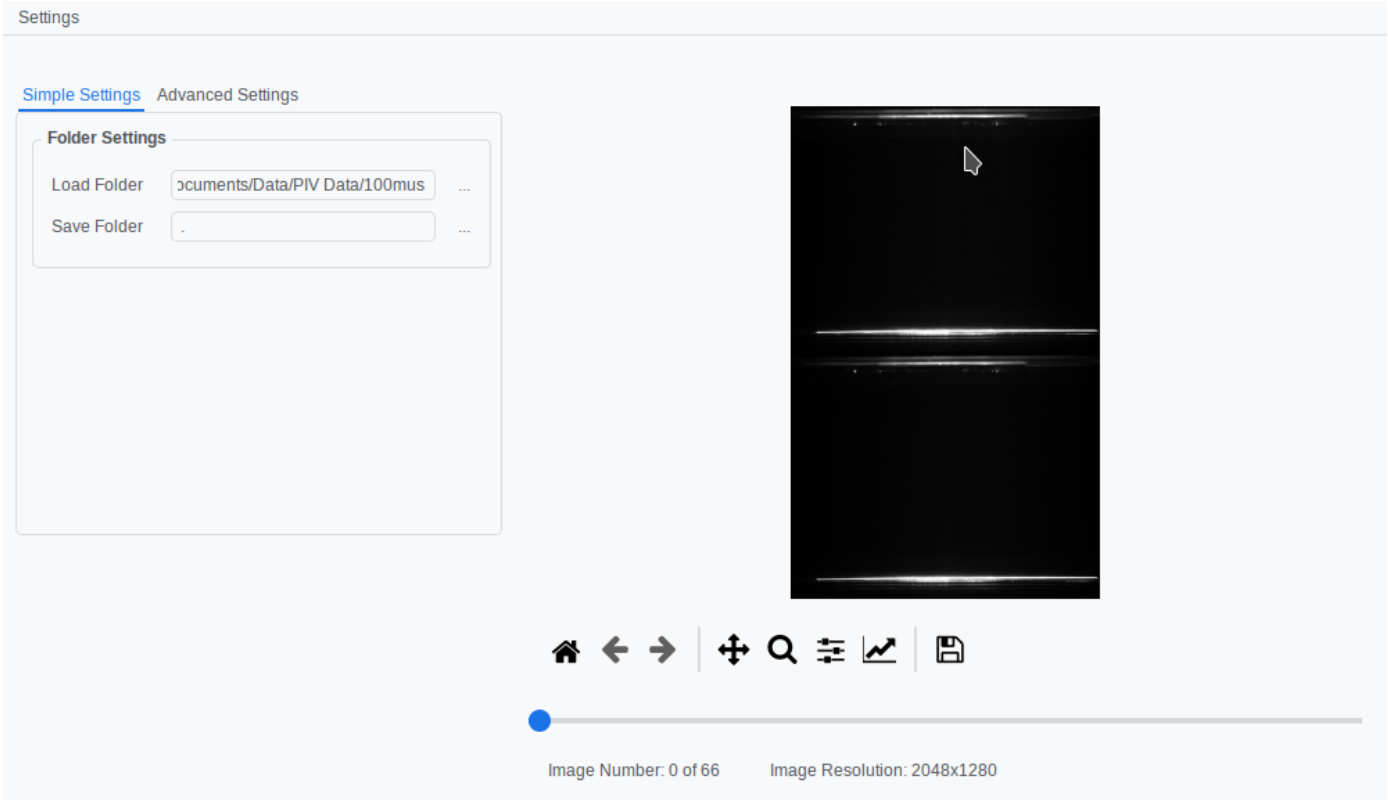
Looking at Images

- Now let's create our functions
  - Next we create the function to update our plot
- This function doesn't take much time to run
  - We try to read the image and then plot using our "canvas" (usual functions for matplotlib apply)
  - Then we draw the figure
  - And update the labels
- Don't forget to connect widgets!
  - (Done correctly in finished file)

```
# #When the scrollbar is changed, the plot will show a new image
def update_image(self, num):
    #Read the image
    try:
        image = imread(self.imageList[num], as_gray=True)

    except:
        return

    #Clear axis - usually not the fastest option
    self.livePlotWidget.cla()

    #Show the image
    self.livePlotWidget.canvas.ax.imshow(image, cmap = 'grey')

    #Turn off the axes
    self.livePlotWidget.canvas.ax.axis('off')

    self.livePlotWidget.canvas.figure.tight_layout()

    #Redraw the figure
    self.livePlotWidget.canvas.draw()

    #Update image number on the label
    self.imageNumberLabel.setText('Image Number: %i of %i'%(num, len(self.imageList)))
```

Looking at Images

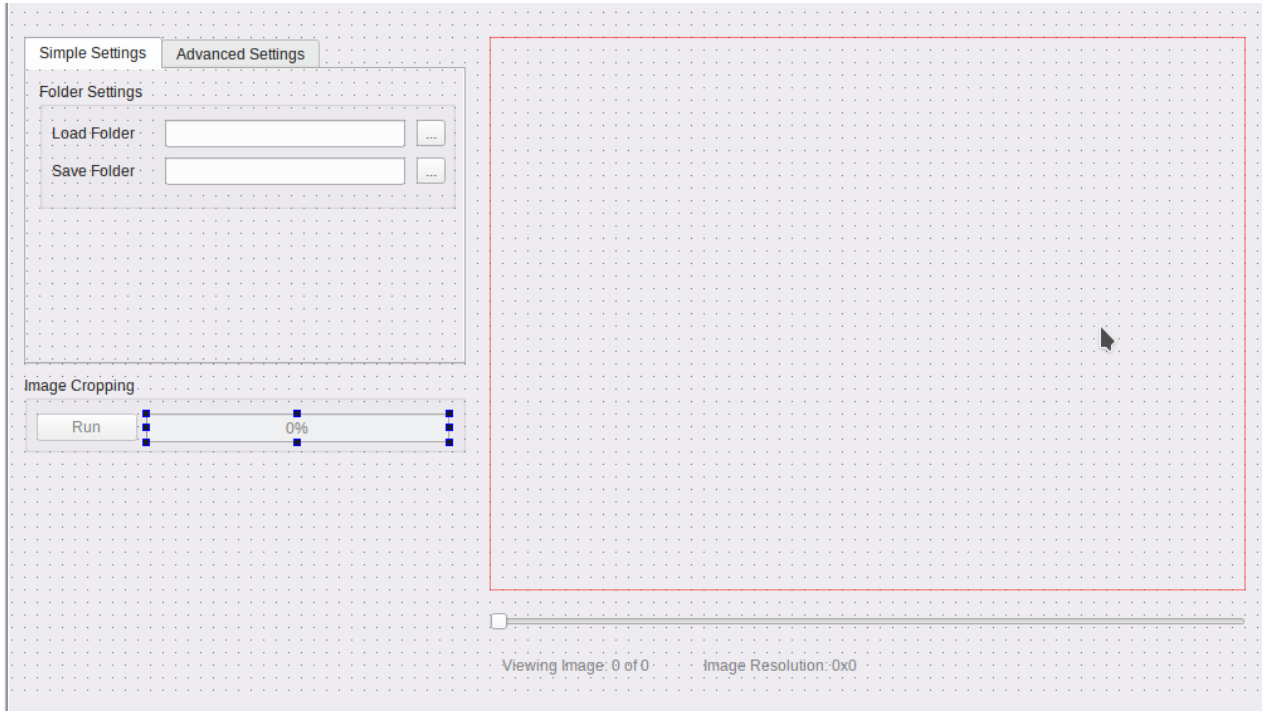
- When we open our image folder, the images should appear!
- We should be able to scroll and see new image pairs



Part 3: A Long Running Function

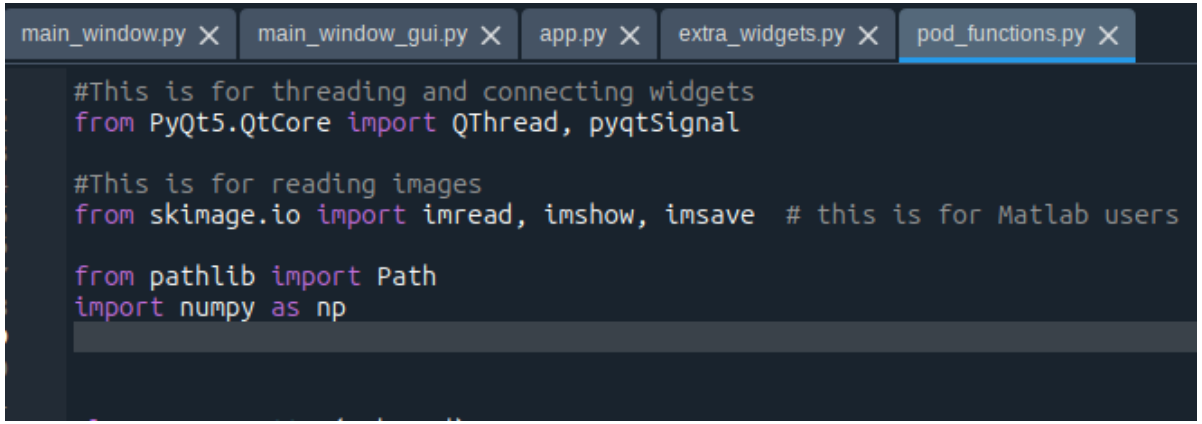
Cutting Image Pairs

- Let's cut our images
  - Saving a bunch of image pairs takes time…
  - This requires us to use threading/multiprocessing so we can still interact with our GUI
- As before… let's create our widgets
  - This time we need a button and a progress bar
  - Set **only the group box** to disabled (we enable them when we load images)



Cutting Image Pairs

- This script will require threading...
  - pyqt has a built in functionality for this
  - We will also illustrate the signal/slot mechanism to update widgets
- Lets make a new file that we will import into the GUI
  - (POD_functions.py)
  - We need to import the QThread object and the pyqtSignal object



Cutting Image Pairs

- We then define a new class
  - It inherits the QThread object to enable threading
- We need to define some signal objects (these connect the GUI output to actions in the thread)
  - They can take in any number/type of object
  - We will connect these to functions in the main GUI

```
class ImageCutter(QThread):
    #This is our signal that takes a number
    updateSignal = pyqtSignal(float)

    #This tells us that the thread is finished and if it succeeded
    finished = pyqtSignal(bool)


    def __init__(self, imageList, saveFolder):
        #Inherit the QThread Class
        super(ImageCutter, self).__init__()

        #We need to know our save folder and the list of our image paths
        self.imageList = imageList
        self.saveFolder = saveFolder / 'cut'

        #Make the folder if it doesn't exist
        self.saveFolder.mkdir(exist_ok = True)
```

Cutting Image Pairs

- We can create our image cutting function
  - Takes some time to cut and save all of the images
- In our loop we update the progress bar with our current progress
- Afterwards, we emit our finished signal if the task finished sucessfully or not
- The "run" function is what gets run when we call thread.start()

```
#This function does the cropping
def cut_images(self):
    #This does the cropping
    try:
        for ii, image in enumerate(self.imageList):
            #Read the image and get the shape
            imageArray = imread(image)
            imageShape = imageArray.shape[0]

            #Create image names for each pair
            imageNameA = self.saveFolder / ('cut_%04d_a.tif'%ii)
            imageNameB = self.saveFolder / ('cut_%04d_b.tif'%ii)

            #Save the image pairs
            imsave(imageNameA, imageArray[0:imageShape//2, :])
            imsave(imageNameB, imageArray[imageShape//2:imageShape, :])

            #Update the progress bar
            self.updateSignal.emit(ii)

        #Makes the progress bar satisfying
        self.updateSignal.emit(len(self.imageList))
        self.finished.emit(True)

    except:
        self.finished.emit(False)
        return

#This is what runs in the thread
def run(self):
    self.cut_images()
```

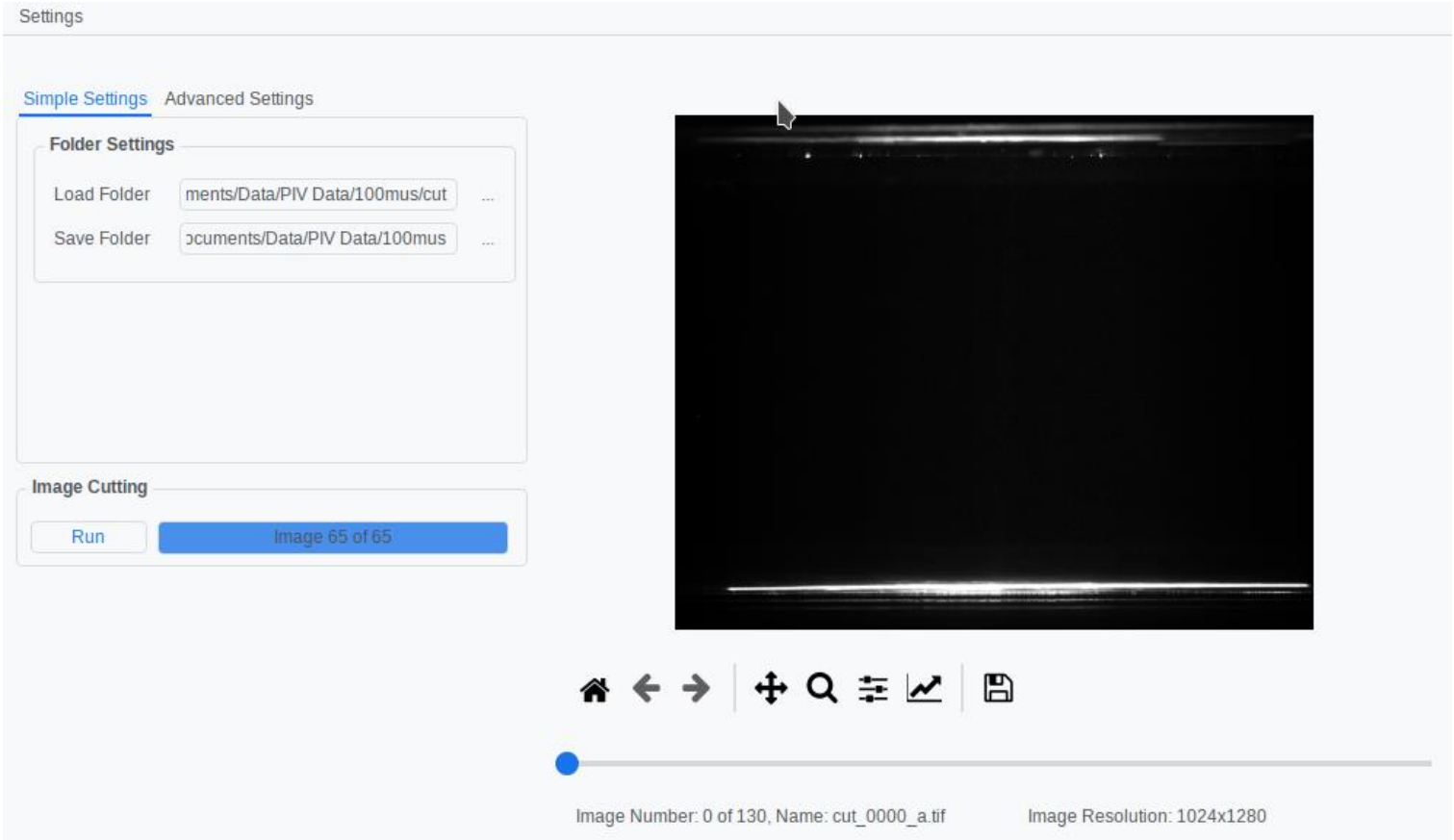
Cutting Image Pairs

- Now in main_window_gui.py
- We create 3 new functions
- cut_images
  - Creates an instance of our new cutter class
  - We connect our signals to the other functions
  - Then we start the thread
- Other functions
  - update_cut_bar updates the progress bar during the loop
  - update_cut_folder will automatically re-load the image folder if the task is successful
- **Don't forget to connect the run button to the cut_images function!**

```
def cut_images(self):
    #Create image cropper class
    self.imageCutter = ImageCutter(self.imageList, self.saveFolder)

    #Connect the signal to updating progress bar
    self.imageCutter.updateSignal.connect(self.update_cut_bar)
    self.imageCutter.finished.connect(self.update_cut_folder)

    #Start thread
    self.imageCutter.start()

def update_cut_bar(self, ii):
    #Set the percentage
    self.cuttingProgressBar.setValue(ii/len(self.imageList)*100)

    #Update the progress
    self.cuttingProgressBar.setFormat('Image %i of %i'%(ii, len(self.imageList)))

def update_cut_folder(self, finished):
    #Update the folder and re-load the images if the crop works
    if finished:
        self.loadFolder = self.imageCutter.saveFolder
        self.loadFolderEdit.setText(str(self.loadFolder))

        self.get_images()
```

Cutting Image Pairs

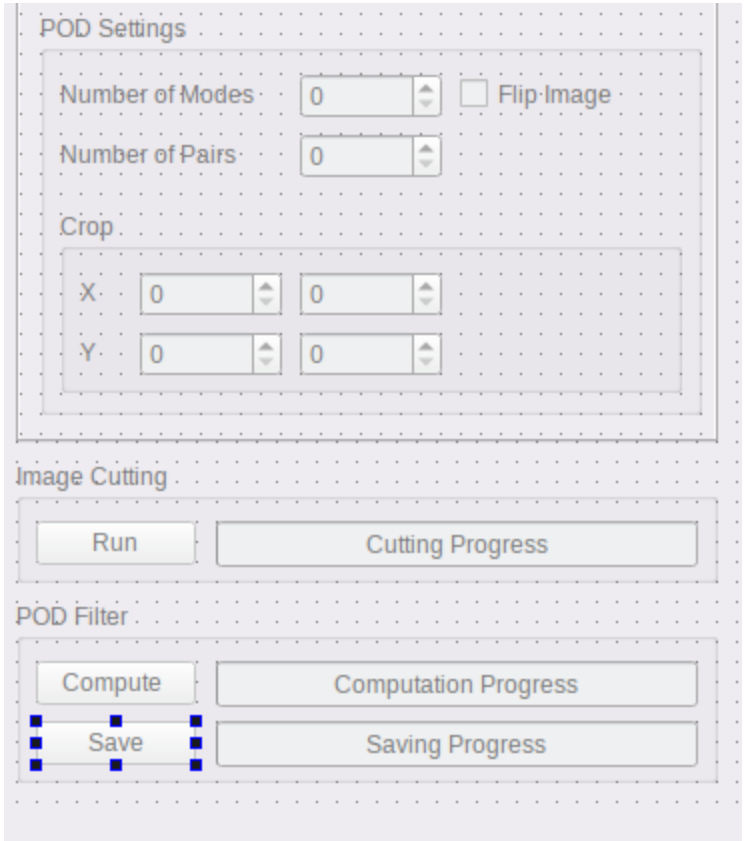
- Now we try this!
- The progress bar should update as our task runs!



Part 4: Implement POD

POD

- Now for the big function!
  - We will use everything we have been doing before to do this
- Of course... let's make our widgets first
  - We have a lot of settings to work through
  - We use the spin box to enter integers
    - "Double spin box exists for doubles..."



POD Thread

- We create our thread object (like before)
  - There are 2 different functions - we switch between by changing a variable in the class
  - This also splits the images by a/b if named correctly

```
class PODRunner(QThread):
    #Our signals to update progress bars
    updateSignal = pyqtSignal(float, str)
    saveSignal = pyqtSignal(float, str)

    #Our signal to indicate if things finished
    finishedComputation = pyqtSignal(bool)
    finishedSaving = pyqtSignal(bool)


    def __init__(self, imageList, saveFolder, settings):
        super(PODRunner, self).__init__()


        #We need to know our save folder and the list of our image paths
        self.imageList = imageList
        self.saveFolder = saveFolder / 'POD_Output'

        #Make the folder if it doesn't exist
        self.saveFolder.mkdir(exist_ok = True)

        #Split the list into a and b
        self.imageAList = []
        self.imageBList = []

        for image in self.imageList:
            if image.name[-5] == 'a':
                self.imageAList.append(image)
            elif image.name[-5] == 'b':
                self.imageBList.append(image)


        #Grab information from settings dictionary
        # Number of modes to remove. If 0, the filter is not active!
        self.nModes = settings['nModes']
        self.nPairs = settings['nPairs']
        self.flipImage = settings['flipImage']

        #Crop list - [X1, X2, Y1, Y2]
        self.cropList = settings['cropList']

        #This variable changes if we want to save/compute the filtered matrix
        self.function = 'compute_matrix'
```

POD Thread (cont)

- Our big function does all of the POD (see machine learning/DDMA lectures)
- We emit a string and percentage for the progress bar
- The projection matrices are the only ones that are stored

```
def compute_filtered_matrix(self):
    #Prepare image matrix and calculate shape
    imInitial = imread(self.imageAList[0])

    #Create image crop
    croppedImage = imInitial[self.cropList[2]:self.cropList[3], self.cropList[0]:self.cropList[1]]

    #Flip the image if we want to
    if self.flipImage:
        croppedImage = np.fliplr(croppedImage)

    #Get the shape of the image
    self.imageShape = croppedImage.shape
    ny, nx = self.imageShape



    #Create matrix to concatenate imasges
    D_a = np.zeros((nx * ny, self.nPairs))  # Initialize the Data matrix for image sequences A.
    D_b = np.zeros((nx * ny, self.nPairs))  # Initialize the Data matrix for image sequences B.

    #Update progress
    self.updateSignal.emit(0, 'Start Processing')

    #Process image pairs and concatenate
    for k in range(0, self.nPairs):
        #Read images
        imA = imread(self.imageAList[k])
        imB = imread(self.imageBList[k])

        #Create image crop
        cropA = imA[self.cropList[2]:self.cropList[3], self.cropList[0]:self.cropList[1]]
        cropB = imB[self.cropList[2]:self.cropList[3], self.cropList[0]:self.cropList[1]]

        #Flip the image if we want to
        if self.flipImage:
            cropA = np.fliplr(cropA)
            cropB = np.fliplr(cropB)

        #Cast to float array
        cropA = np.float64(cropA)  # We work with floating number not integers
        cropB = np.float64(cropB)  # We work with floating number not integers

        # Reshape into a column Vector
        cropA = np.reshape(cropA, ((nx * ny, 1)))
        cropB = np.reshape(cropB, ((nx * ny, 1)))

        #Append to matrix
        D_a[:, k] = cropA[:, 0]
        D_b[:, k] = cropB[:, 0]
```

POD Thread (cont)

- The saving images thread is very simple
- It reshapes the image from the stored projection matrix and saves it

```
def save_images(self):
    (ny, nx) = self.imageShape
    for k in range(0, self.nPairs):
        #Save A images
        imdA = np.copy(self.D_a_filt[:, k])
        imPODA = np.reshape(imdA, ((ny, nx)))
        imPODA[imPODA < 0] = 0  # Things below 0 are treated as zero

        imPODA = np.uint8(imPODA)
        imsave(self.saveFolder / ('POD_Filt_%03d_a.tif'%k), imPODA)

        #Save b images
        imdB = np.copy(self.D_b_filt[:, k])
        imPODB = np.reshape(imdB, ((ny, nx)))
        imPODB[imPODB < 0] = 0  # Things below 0 are treated as zero

        imPODB = np.uint8(imPODB)
        imsave(self.saveFolder / ('POD_Filt_%03d_b.tif'%k), imPODB)

        #Update the signal
        self.saveSignal.emit(k/self.nPairs*100, 'Saving Image Pair %i of %i'%(k, self.nPairs))

    self.saveSignal.emit(100, 'Finished Saving')
    self.finishedSaving.emit(True)
```

POD In the GUI

- After importing the thread object we need to create functions to run the functions for saving/computing
  - In the function we pass the parameters from the boxes to a podRunner object
  - Then the signals are connected to the relevant functions
  - Then we set the function to "compute_matrix"
  - Then we start the thread
- The user can only save images once the POD is computed
  - This is done by enabling/disabling widgets

```
def compute_pod_matrices(self):
    #Collect settings
    settings = {}
    settings['nModes'] = self.podModeBox.value()
    settings['nPairs'] = self.podPairBox.value()
    settings['flipImage'] = self.podFlipImageCheckbox.isChecked()

    #Crop list - [X1, X2, Y1, Y2]
    settings['cropList'] = [self.xCropMinBox.value(),
                            self.xCropMaxBox.value(),
                            self.yCropMinBox.value(),
                            self.yCropMaxBox.value()]

    #Create podRunner object
    self.podRunner = PODRunner(self.imageList, self.saveFolder, settings)

    #Connect signals to functions
    self.podRunner.updateSignal.connect(self.update_pod_bar)
    self.podRunner.saveSignal.connect(self.update_pod_save_bar)
    self.podRunner.finishedComputation.connect(self.on_finished_computing)
    self.podRunner.finishedSaving.connect(self.on_finished_saving)

    #Set the function to compute matrix and start
    self.podRunner.function = 'compute_matrix'
    self.podRunner.start()

def save_pod_images(self):
    #This sets the podRunner to save images and starts the thread
    self.podRunner.function = 'save'
    self.podRunner.start()
```

POD in the GUI (cont)

- The progress bars also need to be updated
- The gui also needs to do something if the saving/computing is successful

```
def on_finished_computing(self, finished):
    if finished:
        self.podSaveButton.setEnabled(True)
        self.podSaveProgress.setEnabled(True)
        self.showComputedImagesCheckbox.setEnabled(True)

def on_finished_saving(self, finished):
    if finished:
        self.loadFolder = self.podRunner.saveFolder
        self.loadFolderEdit.setText(str(self.loadFolder))

        self.get_images()

#Functions to update POD progress bars
def update_pod_bar(self, percent, label):
    #Set the percentage
    self.podProgressBar.setValue(percent)

    #Update the progress
    self.podProgressBar.setFormat(label)

def update_pod_save_bar(self, percent, label):
    #Set the percentage
    self.podSaveProgress.setValue(percent)

    #Update the progress
    self.podSaveProgress.setFormat(label)
```

POD in the GUI (cont)

- We also can edit the update_image function to preview
  - **Bug:** this does not show all of the images, likely to do with num>len(D_a_filt)

```
#Preview images from POD if we want to
if self.showComputedImagesCheckbox.isChecked():
    #Select image pair
    if num%2==1: #even images are b
        image = np.copy(self.podRunner.D_b_filt[:, num])
    else:
        image = np.copy(self.podRunner.D_a_filt[:, num])

    (nx, ny) = self.podRunner.imageShape
    image = np.reshape(image, ((ny, nx)))
    image[image < 0] = 0  # Things below 0 are treated as zero

    image = np.uint8(image)
```

POD in the GUI (cont)

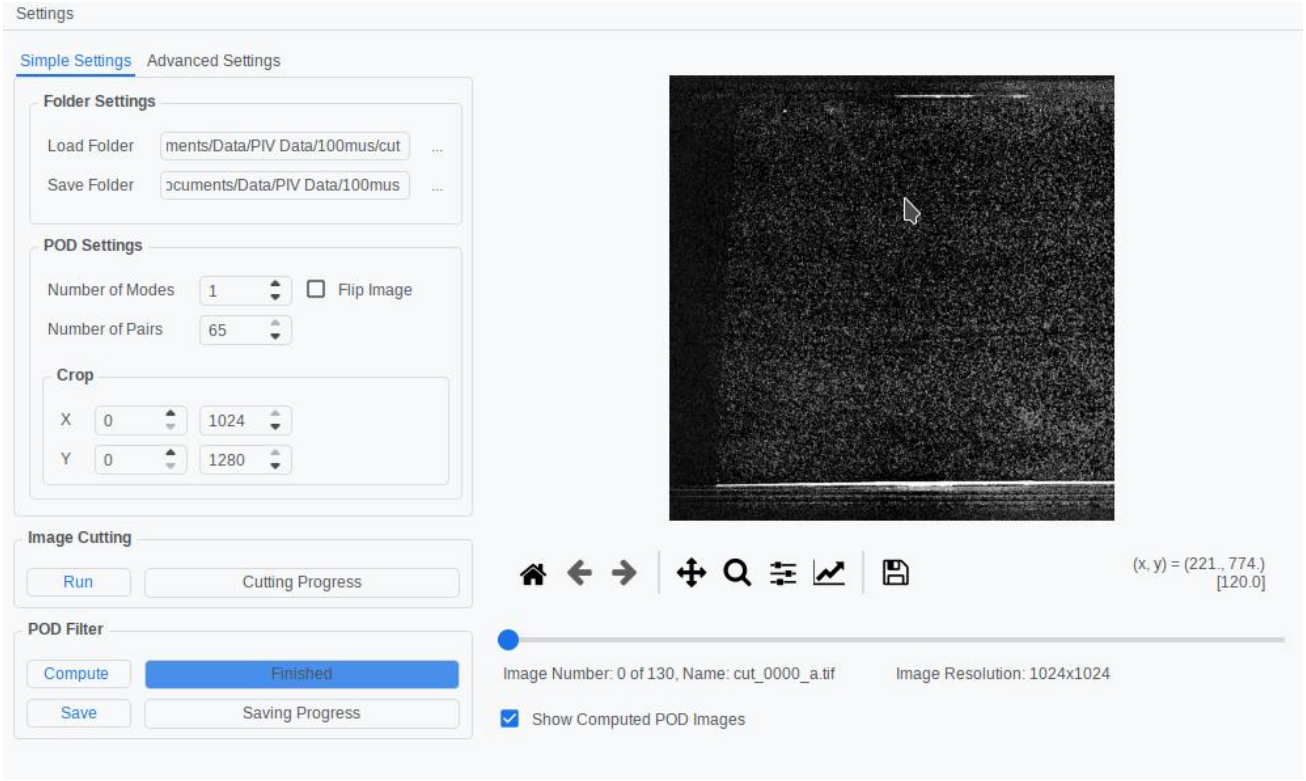
- We need to enable the right widgets when we
- Also need to connect widgets
- Overall, this gets tedious more than complicated

```
#This tells the app the image files in our folder
def get_images(self):
    #List all files in the image folder and sort
    self.imageList = sorted([file for file in self.loadFolder.glob(IMAGE_TYPE)])

    #Update widgets if there are images
    if len(self.imageList) > 0:
        #Set the scroll bar maximum
        self.imageScrollBar.setMaximum(len(self.imageList)-1)

        #Enable the widgets
        self.imageScrollBar.setEnabled(True)
        self.imageNumberLabel.setEnabled(True)
        self.imageResolutionLabel.setEnabled(True)

        #Image crop group
        self.cuttingGroup.setEnabled(True)
        self.cuttingProgressBar.setEnabled(True)
        self.cuttingRunButton.setEnabled(True)

        #POD Group
        self.podFilterGroup.setEnabled(True)
        self.podRunButton.setEnabled(True)
        self.podProgressBar.setEnabled(True)
        self.podSettingsBox.setEnabled(True)

        #Set maximum number of pairs
```

```
#POD Buttons
self.podRunButton.clicked.connect(self.compute_pod_matrices)
self.podSaveButton.clicked.connect(self.save_pod_images)
self.showComputedImagesCheckbox.clicked.connect(self.update_image)
```

POD Finished (mostly)

- This is what it looks like when it is done!
- There are some bugs/features that are left as an exercise to the reader
  - **Feature:** implement real-time feedback on cropping images
  - **Feature:** implement naming folders/a "workspace"
  - **Bug:** previewing POD images does not show both "a" and "b" images
- You can preview the POD computation before saving and save the images



Extra Slides

Appendix: Installing Requirements

- Create a new conda environment (conda create –n envname python=3.9)
  - Use python3.9 for compatibility
- Install pip "conda install pip"
- The main folder has a requirements.txt file
  - Install using "pip install -r requirements.txt"

Python Frameworks

- There are many frameworks available to create GUIs
  - PyQt and tkinter are the biggest
- PyQt is preferred
  - Has a nice widget editor
  - Better multiprocessing support
  - Compiled in c++
- PySide vs PyQt
  - Both use Qt